A micro-computer implementation of an interactive functional programming system

Wei Zi Chu

INRIA

CENTRE DE RENNES
IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(3) 954 90 20

Rapports de Recherche

N° 277

A MICRO-COMPUTER IMPLEMENTATION OF AN INTERACTIVE FUNCTIONAL PROGRAMMING SYSTEM

WEI ZI CHU

Mars 1984

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Publication Interne n°219
Janvier 1984
22 pages

A MICRO-COMPUTER IMPLEMENTATION OF AN INTERACTIVE FUNCTIONAL PROGRAMMING SYSTEM

Wei Zi Chu
IRISA/INRIA *
Campus de Beaulieu
35042 RENNES CEDEX

and

INSTITUTE OF MATHEMATICS, ACADEMIA
SINICA

Résumé : Ce papier présente une mise en oeuvre du langage fonctionnel (FP) sur le micro-ordinateur MICRAL-REE. Nous décrivons la structure du système et l'organisation des données basée sur des arbres binaires. Les résultats principaux sont exhibés et discutés. Les améliorations possibles et le développement futur du système sont évoqués en conclusion.

Abstract : This paper presents a practical micro-computer implementation of the functional programming language (FP). The program structure and data organization based on binary trees are described. The main results are exhibited and discussed, and further developments are quickly sketched.

*) This work was done in 1983 at IRISA/INRIA (FRANCE).

Key Words

functional programming systems (FP), interpretation of FP, software construction, micro-computer workstation.

It's no doubt that the Turing Award Lecture given by J. Backus [1] six years ago was a challenge to the conventional computer architecture and programming languages of Von Neumann style. From then on, more and more experts believe that the design of the next generation computers will be influenced significantly by functional programming [7]; and functional programming languages will play an important role in the development of programming languages of the 80's [6]. But just as said by J. Backus himself, only when those models of computing systems in new style and their applicative languages have proved their superiority over conventional languages will we have the economic basis to develop the new kind of computer that can best implement them, only then, will we be able to fully

utilize large-scale integrated circuits in a computer design.

Based on this opinion and for providing an educational tool of programming for the beginners as well, an interactive functional programming working system called FPW has recently been implemented on a micro-computer----MICRAL-REE by using of PASCAL. In the following we will describe briefly the schema used in FPW implementation.

1. Overview of FP

FP (Functional Programming) as proposed by J. Backus consists of five basic parts:

p1. A set of objects. It includes the elementary elements so called atoms (numbers, strings of characters and so on) and a variety of sequences with the components which are still objects.

p2. A set of primitive functions.

p3. A set of functional forms. With which complex functional expressions can be built up by using of known functions.

p4. A set of definitions. With which new functions, especially the recursive functions and functional equations, can be defined by using functional expressions.

p5. A unique operation----the application of any of the functions formed as above to an appropriate object.

With this idea, a FP program is just a simple functional expression which maps objects (as the data-in) to other objects (as the data-out).

2. About MICRAL

Our FPW system is implemented in the micro-computer MICRAL-REE[4].

MICRAL-REE is a french personal computer widely used as a teaching instrument in universities. The Central Processing Unit is constructed around a 16 bits micro-processor driven by a 5 MHz clock and possess 22 interrupt levels. An I/O processor assures the direct access function from disc readers to the memory. The interface system includes a disc controller, a parallel export device and four V24 channels of 75-19200 bauds, etc. There are a central memory in 256 k bytes (may be extended to 1024 k bytes), a PROM in 4 k bytes (which includes an auto-diagnotor, a test monitor and a disc-system loader) and two readers of floppy disc in 320 k bytes. The basic software contains the operating system CP/M-86, an assembler, a PASCAL compiler. The CP/M-86 is a multi-user and multi-tasking monitor, it assures the management of disc files and the dialogue between central processor and peripheral units.

3. The implementation of FP

There is a synthesis for the elementary principles for implementing functional languages in document [5]. Now we take a practical implementation about the FP system in the micro-computer into account.

3.1 The language FPW

Atoms of FPW include alphabetic strings (identifiers and numbers) acceptable to the machine. Real numbers have been chosen for a better applicability of the system (for example for mathematicians). The objects may be the simple atoms as well as all kinds of sequences. For example, the empty sequence < > and the following pair of matrices

<<<A11, -3.4E+5>, <0.3, +81.2>>, <<-5, B12>, <0, 7E-2>>>

are objects of the language.

FPW accepts all of the most basic functions proposed by J. Backus and others [1] [8] as its set of primitive functions: 1,2,...(selectors), TL (tail), ID (identity), +, -, *, / (arithmetic operations), AND, OR, NOT (logic operations), EQ, GT, LT (comparisons), ATOM, NUMB, NULL (judgements), REV (reverse), TRANS (transposition), APNDL, APNDR (append), DISTL, DISTR (distribution). Furthermore, some arithmetic functions ODD, SQRT, EXP, LN, SIN, COS, ARCTAN have been counted as well. In general, all of other functions can be obtained by composition of the above mentioned functions. For example, the minus-one function, the rotate-left function and the IOTA function which produces a sequence of successive integers starting from 1 can be expressed respectively as follows:

SB1 = - o [ID, $1]

ROTL = APNDR o [TL,1]

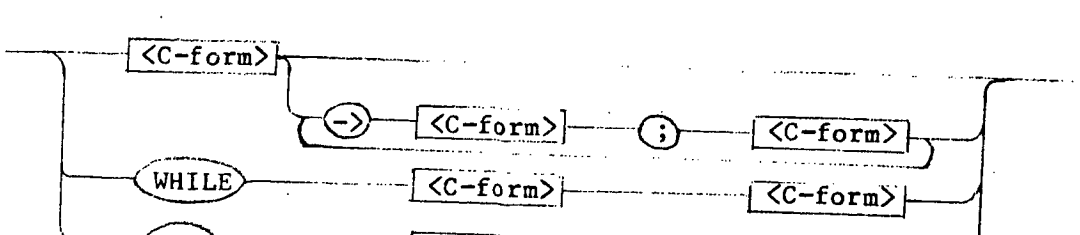
IOTA = (WHILE GT o [1, $1] APNDL o [SB1 o 1, ID]) o [ID]

Where $1 is used to denote the constant 1.

FPW contains all the functional forms proposed by J. Backus. For avoiding unnecessary parentheses and making the writing of expressions more elegant, the syntax of functional forms has been modified as

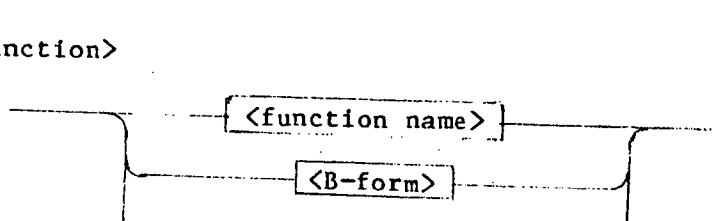
follows:

<form>



<C-form>

<function>



<B-form>

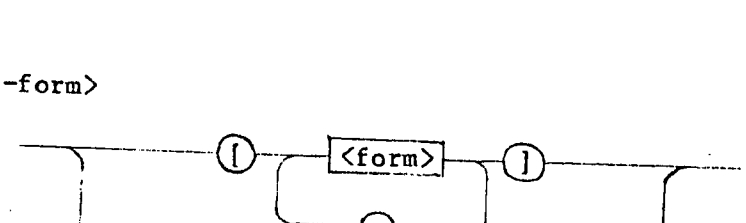


Fig.1 The syntax of functional forms

Where \ is used to denote 'right insert'. @ is used to denote 'apply to all'. We assume that a series of compositions are realized in turn

from left to right except using parentheses to change this order. In this way, F1 o F2 o F3 o F4 is equivalent to ((F1 o F2) o F3) o F4.

By the above syntax, we in fact provide an order of the priority for symbols of functional forms and other separators, that is

1st level: $, \, @, [ ], ( )

2nd level: o

3rd level: WHILE, BU, -> ;

4th level: ,

Finally, it is evident that all functions included in such a system are strict.

3.2 Main elements of FPW

FPW is a interactive interpreter. The facilities of this system as seen by the users are:

f1. Defining new functions.

f2. Applying any defined function to a determinate object to get a result.

f3. Saving functions in a library.

f4. Listing all functions which can be used at that time, or listing the defined expression of a function.

f5. Deleting a saved function.

f6. Renaming a defined function.

Fig.2 summarizes the FPW features. The total FPW system can be divided into five basic modules and some auxiliary parts. These modules are named respectively as analysis module, interpreter module, save module, list module and delete module.

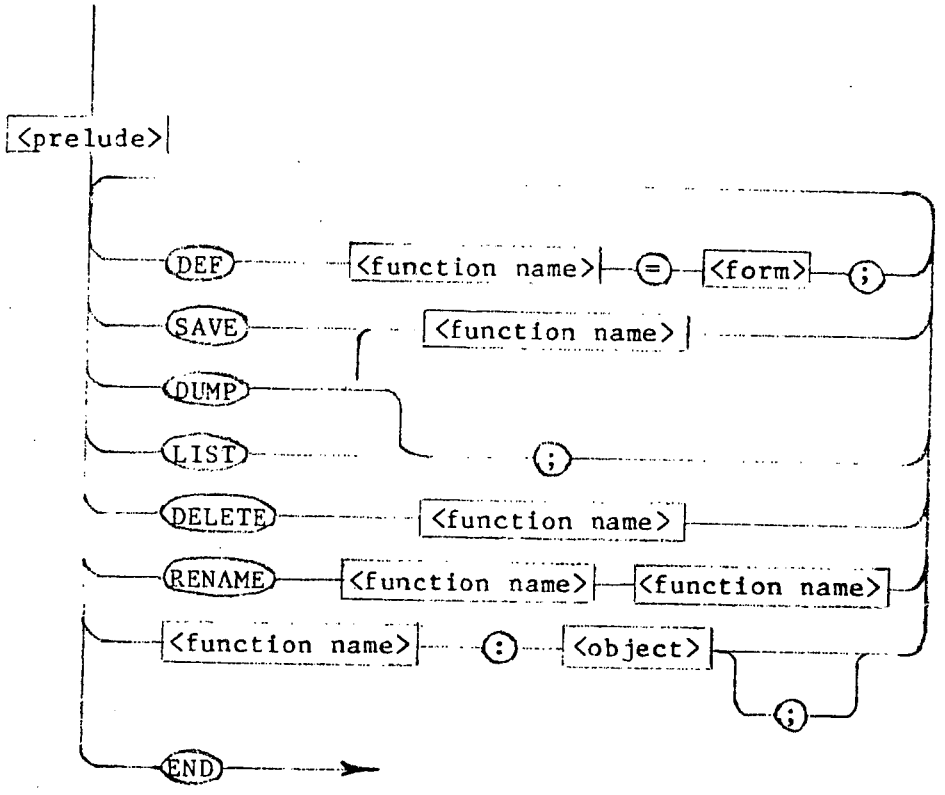


Fig.2 Survey of FPW

3.2.1 Analysis module.

Beside the parsing of the input data, the key-job of this module is to transform each function definition into a binary-tree in inner denotation (see the subsection 3.3). Different functional forms are reflected by the difference between nodes of the corresponding trees. Complex functions are characterized by trees with subtrees in different hierarchies. For example, the functions calculating inner product and factorial, IP = \+ o @* o TRANS and FAC = EQO -> $1; * o [ID, FAC o SB1], have the following binary-tree representations.

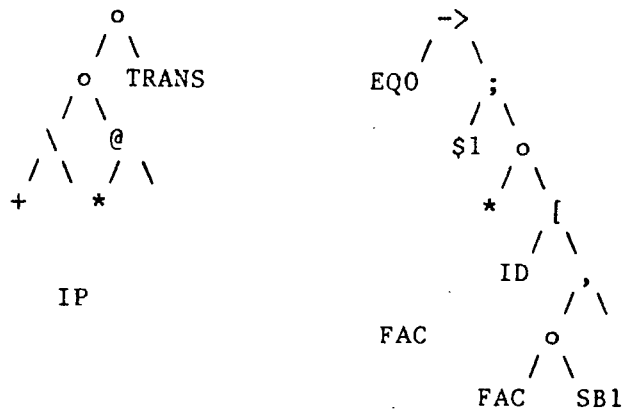


Fig.3 The binary-tree representation of functions

The leaf nodes are primitive functions, objects or ready-defined functions like EQ0 and SB1, they may even be the recursively defined functions themselves like FAC. For certain functional form, such as \ (insert) and @ (apply to all), the binary-tree has an empty branch.

3.2.2 Interpreter module.

It consists of two parts, a package to deal with primitive functions and a program to apply binary-trees. The latter is used to find the binary-tree of an invoked function and to apply this binary-tree to the given object by doing some semantic processing. The interpretation of a binary-tree consists of that of its two subtrees: if a subtree is a leaf node, then apply the corresponding function, otherwise the subtree can be further biparted. The order of interpreting left and right subtrees should be in accordance with the way of constructing binary-trees (the latter is in the natural order of writing). As we can see in a concrete instance in Fig.4, different functional forms may have different evaluation orders. For example, the order is from right to left for composite forms,

while from left to right for conditional forms.

step	operation	effect
0	execute the tree of FAC	FAC:6
1	execute the left-subtree of conditional form	EQ0:6=F
2	execute the right-tree	
3	due to EQ0:6=F, execute its right-subtree	FAC:6=*o[ID, FACoSB1]:6
4	execute right-subtree of composite form	[ID,FACo SB1]:6
5	execute left-subtree of construction form	ID:6=6
6	execute right-subtree of construction form	FACoSB1:6
7	execute right-subtree of composite form	SB1:6=5
8	execute left-subtree of composite form	FAC:5
	.	.
	.	.
	.	.
n-1	.	FAC:5=120
n	execute left-subtree of composite form	FAC:6=*:<6,120> =720

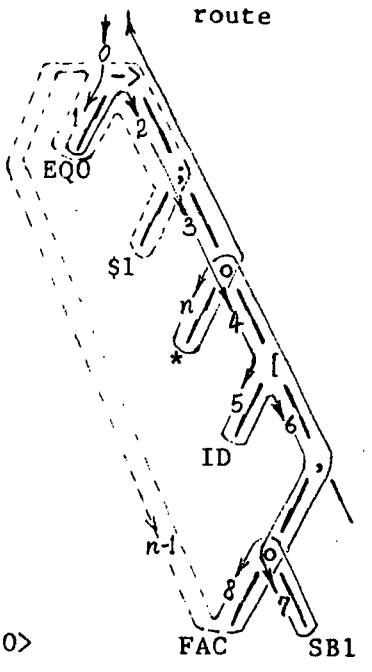


Fig.4 The interpretation of the binary-tree of FAC

The communications between the first two modules is done through data tables described in the next subsection.

3.2.3 Save module.

In order to avoid frequent data exchange between main and secondary storage, this module is divided into two parts: a move program and a DUMP procedure. The former deal with the movement of binary-tree of a function needed from temporary data tables to normal data tables and transform the representations of constants included in that function (see next subsection). The data of a saved function need not be moved into secondary storage until a DUMP command be called.

3.2.4 List module.

It allows the user to list all functions in his library (saved and defined a moment ago) and to reproduce the original functions from their binary-tree representations.

3.2.5 Delete module.

It allows users to be able to delete or to rename a function existing in his library. When deleting a saved function, the system collect back the unuseful storage in normal data tables.

3.3 Data organization in FPW system

It is easy to represent data with recursive structure by use of PASCAL records, so FPW objects can be represented with PASCAL linked lists. Since FPW objects might be distinguished as three types (character strings, numbers and sequences), they are represented as a pair <tag, pointer>: The pointer points to the real position of that object, and the tag is the type of that object body. We denote the types of identifiers, numbers and sequences with I, N and S respectively, then the sequence X=<<AB1, <>, 5>, <3.14, R>> can be represented as in Fig.5.

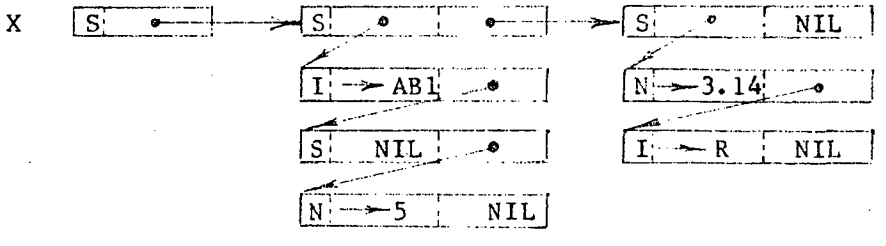


Fig.5 The representation of an object

Unfortunately, the PASCAL linked lists can not be used to record tree structures of FPW functions. This is because PASCAL pointers can't do arithmetical operations with integers, but some of these operations are needed when we save a function and want to move its binary-tree. An alternative choice is to build a table by an array to describe the tree structure of functions. In order to save storage, we set up two tables, the F-table and the T-table, they are illustrated in Fig.6 by the function FAC.

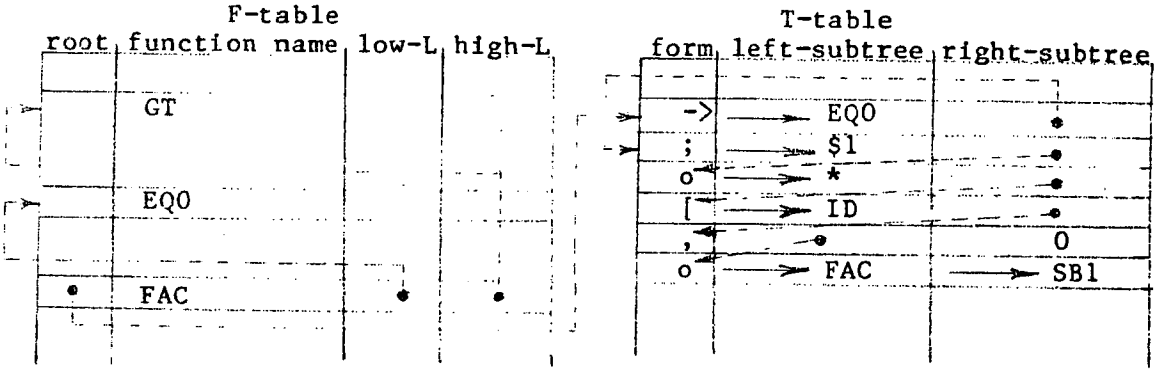


Fig.6 The tree structure tables in FPW system

The bidirectionnal link in the F-table is used to increase the search speed.

It is necessary to consider the problem of saving constants which appear in a function definition. There is no way to record constants in linked list form into secondary storage directly, because pointers are dependent on the current state of the main storage. Thus another kind of object, object in table form, is needed, where array indices are used to represent link relations instead of pointers. Therefore there exist two other tables, the O-table and the S-table. If the object X mentioned above is a constant needed to save, it can be represented as follows.

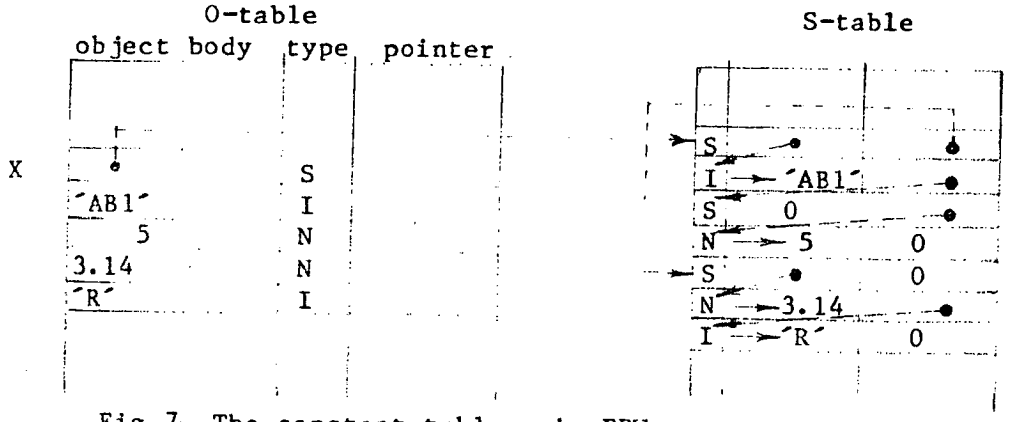


Fig.7 The constant tables in FPW system

The conversion of objects in linked list form to that in table form is done by save module whenever necessary. However, it is inconvenient for the interpretation of functions to use two kinds of object representations. So, when FPW begins to run, all objects in table form would be converted into linked list form one. There is a column in the O-table which is used to record the actual position of the object body after the conversion. With this position the interpreter can access the object body.

3.4 Debugging aids

It is important to attach some debugging facilities to a interpreter. Besides the static error examination, four switches may be put up into FPW system. One of them is available to users. When putting on it the system will produce the sequence of invoked functions and of their operands. Fig.8 provides an example. By doing so users can find the information of dynamic errors easily. The other switches are available to maintainers. They allow to display the execution track of the system and contents of the four data tables

mentioned above.

```
  FAC:4
EQO   4|ID    4|EQ  <4,0>|ID    4|SB1   4|ID    4|-   <4,1>|FAC   3|
EQO   3|ID    3|EQ  <3,0>|ID    3|SB1   3|ID    3|-   <3,1>|FAC   2|
EQO   2|ID    2|EQ  <2,0>|ID    2|SB1   2|ID    2|-   <2,1>|FAC   1|
EQO   1|ID    1|EQ  <1,0>|ID    1|SB1   1|ID    1|-   <1,1>|FAC   0|
EQO   0|ID    0|EQ  <0,0>|*  <1,1>|*  <2,1>|*  <3,2>|*  <4,6>|
  FAC   :4
    ==24
```

Fig.8 The process of applying a function

3.5 Garbage collecting

As the design FPW system is for providing a simple tool to educate to FP programming, the target of FPW garbage collecting is not in getting the posibility of processing complex problem, but in the guaranty of a continuous manipulation with this system. FPW adopts a simple technique: to find the interface between FPW system and the running procedure NEW of PASCAL system which allots the dynamic data locations. After each execution of applying a function to a object, FPW makes the pointer of dynamic data heap to return back to the position it possessed before this application. By doing so, all data which represent constants occurring in each function definitions are always at the head of dynamic data heap, and there is a free space for the application of functions.

3.6 Expandability and independence.

FPW uses four compatible files to record the contents of the four data tables. These are the initial values of those tables for the next run and will be changed if the user saves new functions. Once a user-defined function

is saved into files, it can be used like a primitive function. In this way, FPW can get its facilities extended from time to time. The nature of micro-computers makes it possible that each user can have his own system under his own control.

3.7 Transportability

Care has been taken to make the implementation of FPW independent of particular machine during the design phase. Almost all of the machine-dependent features have been collected and put into the constant declaration part. However, the contents of four files are dependent on the sizes of four data tables which are in turn dependent on the memory size of the particular machine and on user needs. For the current version four to five hundred function definitions can be put into these tables. When the sizes of these tables change, the four files have to be reconstructed.

4. Results

We have spent about two man-months in the implementation of FPW system. Half of our time was spent in designing the FPW specification and its implementation schema, including system structure and data representation details, other time was spent in coding and testing. After that we have made some improvement and modification intermittently. FPW consists of approximately 2600 lines of PASCAL code (about 52k bytes of object code and 3k bytes of data). The testing has been done without many hitches. If there is an error in a FP program, it is easy to be found with the help of messages given by the system. These in turn show the advantages of the functional

programming[2],[6]:

1. Languages without variables eliminate side effects during program execution.

2. Due to the static structure of a program conforms with its dynamic execution, program reliability has been improved.

3. The hierarchical structure built with composition forms adds clarity to programs.

4. Extremely simple semantics reduced the implementation complexity and the amount of errors made in programming as well.

Perhaps, for the users, the following two aspects are very useful. Firstly, programming with FP, complex problems can be expressed hierarchically in simple notations and in easy-to-understand formats just like using elementary algebra. Since FP programs are more like mathematical expressions, it is more easy to learn programming with FP languages than with any other high level programming language. Secondly, as we have said, there is an expandability of our system, this means that the method of FP provides a posibility of building, storing and using software components. Any predefined function can become a build-in block to the future programs. This might be a progress in software engineering.

However, at least on Von Neumann computers, the advantages of functional programming are at the price of a loss of running efficiency and storage space.

Anyway, our implementation of FP is only a try, but a few enlightements have been gained. It have been predicted that in any implementation of a FP system on a Von Neumann machine the low efficiency will be the main trouble.

Our experience shows that the ability of dealing with recursion is also an greater problem and that the powerful mechanism of garbage collecting are quite necessary.

Since FP is a kind of languages without variables, programming were done in function level, when running a calculating program, there is no variables for access instead some functions (procedures with parameters of input and output), such as identity and selectors, have to be invoked. Similarly, simple operations (such as arithmetic operations, comparisons) are all become to invoke functions. If we calculate (X-1) * X, programming in Von Neumann languages is very simple, it just like

```
GET     X
-       (1)
*       X
```

there are only tree instructions. But if we design with FP, things are more complicated. We have to write it as

```
* o [ - o [ ID, $1 ], ID ] : X
```

and many procedures would be invoked:

```
call procedure          X1= ID:X
produce new sequence    X2=<X1, 1>
call procedure          X3= -:X2
call procedure          X4= ID:X
produce new sequence    X5=<X3, X4>
call procedure              *:X5
```

Moreover, the access of a component of a sequence (dispersed variable) is generally more complex than that of an indexed variable (continuous variable). All of these might be the main causes of the lower efficiency of a

FP program executed in a Von Neumann machine. How can we change such a weakness of languages without variables is just a great problem. But we can expect that optimization of the implementation of primitive functions would reduce the efficiency loss of FP program executions.

Now, in our system, the efficiency lost is not yet so obvious at least for problems which are not very complex. The following table list some macroscopic practical figures.

function	operation	run time (seconds)
FAC=EQ0 -> $1; * o [ID, FAC o SB1]	FAC : 15	< 1
	FACT: 15	< 1
	FACTOR:15	< 1
FACT=2o(WHILE GTo[1,$1] [SB1o1,*])o[ID,$1]	FAC : 97	3.5
	FACT: 97	3.5
	FACTOR:97	3.5
FACTOR=* o IOTA	FACT:170	6
	FACTOR:170	6.5
FIBONAC=2 o (WHILE GTo[1,$0] [SB1o1,+oTL,2]) o [ID, $0, $1]	FIBONAC:20	< 1
	FIBONAC:100	3
	FIBONAC:750	18
MM=@@IP o @DISTL o DISTR o [1, TRANS o 2]	4*4dimension	2
	8*8dimension	13
	9*10dimension	18

Fig.10 Examples of run times for classical programs

These results are acceptable if we compare them with two other figures: using a PASCAL program to calculate factorial 170, the running time is about 3 seconds; and the time for displaying the value of factorial 170 is about 12 seconds.

However, the implementation of FP requires the machine to be very powerful in dealing with recursion (the same requirement for the writing tool of a FP system). In our current implementation schema, there would be several dozens

of recursion levels even in computing a very simple problem. Some practical numbers list in the following table.

operation	number of recursion levels
FAC : 3	15
FAC : 13	55
product of two 3-dimension vectors	5
product of two 3*3-dimension matrices	10
FIB : 3	13

Fig.11 Exemples of number of recursion levels

(where FIB= LE1 -> ID; + o [FIB o SB1, FIB o SB2] and SB2 = - o [ID, $2]) As for each recursion level, there are still a few complicated procedure calls. Thus, for example, the calculation of FAC:97 has already occupied a recursive stack of about 40 k bytes. It seems that the conventional Von Neumann computers can not afford such powerful recursion requirement. So the ability of a FP system like FPW is still quite limited. Although by extending of the recursion stack things can be improved , but this is not at all the solution for FP, because such a extending is limited and the number of recursion levels goes higher very quickly as the problem turns more complicated or as the magnitude of data turns sufficiently large.

As the other functional programming languages, a FP program generally needs to use a great number of dynamic data, so the problem of garbage collection is very important. In FPW our simple technique of garbage collection is valid for manipulating repeatedly. But this is not enough. We have found that a dynamic data heap of 64 k bytes which is the largest size provided by the machine is not enough to calculate the fibonacci number of 14 by use of the function FIB. A means of garbage collection while applying a function has to be taken into account. There are many methods. But some

features of FP programs can be seen:

1) The FP functions defined with composition have a fine hierarchical structure. We can know that after applying each composite component only the final result is useful, other data locations can all be deleted.

2) In each hierarchical level of a composite form, it is not easy to know which location is still useful or not.

3) Due to FP sequences (objects) are defined recursively, to collect back a object by locations would need to use recursion, this will increase the number of recursion levels and decrease the efficiency in running time.

So, it seems that the best method to collect garbages is still the collecting in batches by means of the interface between systems as we have done after each application of functions in FPW, instead of collecting each unuseful location in turn. With doing so the efficiency loss can be decreased.

Program transformation makes us to be able to evaluate some problems which seems impossible in the original forms. Two examples would show this. As we have pointed out: with FAC only those factorials of the number less than 97 can be evaluated by using a recursion stack of 40 k bytes. But with the variants FACT or FACTOR we can evaluate factorials of all integers until the result is overflow in this machine. Similarly, by using of a dynamic data heap in 64 k bytes, we can't compute the fibonacci numbers of integers which are greater than 13 with FIB, but all of the fibonacci numbers of integers lower than 752 can be evaluated with FIBONAC. So program transformation would be useful to solve the difficulty in some calculations. Fortunately, the method of program algebra helps us to transform FP programs easily. Some work

about program transformation has been done[1],[3]. Therefore, combining with the facilities of it, the system of implement FP will be more powerful.

Functional programming has indeed shown some advantages and the approach of program algebra proposed by J. Backus makes computer science and modern algebra combined together. If new techniques of computer architecture can help it to avoid the encountered difficulties, the FP will show more its power.

Acknowledgement

This paper is a summary of my recent work done at the group led by Prof. J. P. Banatre in Institute IRISA-INRIA, Rennes in France. My thanks are due to J.P. Banatre and D. Le Metayer, they put forward the basic requirement to FPW and suggested to implement it interpretatively by using of the technique of building trees, their support during the period was valuable.

References

[1] J. W. Backus, Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. CACM 21.8 (Aug. 1978).

[2] J. W. Backus, The algebra of functional programs, function level resoning, linear equations and extended definitions. In LNCS,Vol 107, New York, 1981.

[3] J. P. Banatre, D. Le Metayer, An interactive system for the construction of parallel programs. Working note, IRISA-INRIA, FRANCE, Oct.

1982.

[4] J. Barre, F. Jorgensen, Manuel d'utilisation du MICRAL 90 de R2E, Universite de Rennes I, Fevrier, 1983.

[5] D. Le Metayer, Les langages fonctionnels: caracteristiques, utilisation et mise en oeuvre. Research Report, INRIA, n231, Juin, 1983.

[6] R. Q. Lu, The development of higher level languages (2), Computer research and development. (chinese), May, 1982.

[7] P. C. Treleaven, Computer architecture for functional programming. Advanced course on functional programming and its applications. University of Newcastle (UK), July, 1981.

[8] J. H. Williams, On the development of the algebra of functional programs. ACM Transactions on Programming Languages and Systems, Vol 4, No. 4 (Oct. 1982).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique